SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 00(00), 001–022 (April 1997)

Workplace Microkernel and OS: A Case Study

Brett D. Fleisch & Mark Allan A. Co
{brett|marcus}@cs.ucr.edu

Department of Computer Science, University of California, Riverside, CA 92521, U.S.A.

SUMMARY

IBM's Microkernel, named Workplace OS microkernel [17], was the core component of Workplace OS, a portable successor of OS/2. The basic premise of Workplace OS work was: 1) IBM would adopt and improve the CMU Mach 3.0 microkernel for use on PDAs, the desktop, workstations, and massively parallel machines, and 2) that several operating system personalities would execute on the microkernel platform concurrently. This architecture would allow users to switch between applications written for different operating systems while IBM would also benefit by having one common platform for all product lines. The goals of the microkernel and the technical features of design are described in this paper. We also present lessons that may benefit future projects with similar goals.

KEY WORDS Operating Systems, Microkernels, Workplace OS

Introduction

IBM's Microkernel, named Workplace OS microkernel [17], was the core component of Workplace OS, a portable successor of OS/2. The basic premise of Workplace OS work was: 1) IBM would adopt and improve the CMU Mach 3.0 microkernel for use on PDAs, the desktop, workstations, and massively parallel machines, and 2) that several operating system personalities would execute on the microkernel platform concurrently. This architecture would provide users flexibility as they switched between applications written for different operating systems. Designed in cooperation with the OSF Research Institute, the IBM microkernel was derived from the Carnegie-Mellon's (CMU) Mach 3.0 microkernel code. IBM envisioned it would benefit from significant cost savings by having one common platform for all product lines.

The Workplace OS project represents one of the most significant operating systems software investments of all time. IBM spent nearly $2 billion on the Workplace project internally or .6% of revenues over the five year span when IBM generated $325 billion in revenues. IBM's strategy for the microkernel required three important components: machine independence, multiple personalities, and concurrent operation of personalities. Each of these aspects was heavily emphasized in the marketing of this new comprehensive IBM strategy. Further, IBM committed significant resources to Workplace OS as a strategic direction for the entire company much in the same way as it had done for the IBM/360 project in the 1960s. Considerable time, programming, marketing, money, and licensing efforts went into Workplace. Despite this, cost overruns, schedule overruns,

0000–0000/00/000000–00$00.00

Received 20th *April 1997*
Revised 23rd *September 1997*

and technical problems plagued the project. Eventually, these issues led to the product's failure and the product was dropped.

In this paper, we analyze one of the most expensive operating systems failures in history from an outsider's perspective. By examining the vendor source code, documentation, and by contacting IBM to supplement the assertions made in this paper, we provide insights into the technical enhancements and improvements over Mach 3.0 that defined the IBM microkernel. We also examine personality development and project history by tracing corporate press announcements, published articles, web information, and adding our own software engineering insights. Finally, we summarize the lessons learned from Workplace.

The next section outlines the new components in the IBM microkernel. A design overview of the microkernel can be found in the Design section. The main features of the microkernel are covered in six subsequent sections: IPC, Task and Thread Management, Power Management, Virtual Memory Management, Logical Clocks for Real-Time Support, and Security. Finally, we present an evaluation of the microkernel, a brief project history, a list of lessons we learned, and our conclusions.

New Features in the IBM microkernel

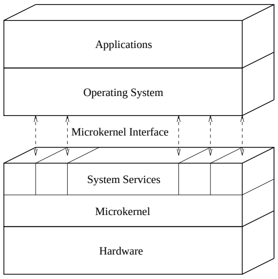

Figure 1: A typical microkernel structure

A *microkernel* is a subsystem that provides the core functions from which other operating system services are built. The microkernel is a layer between the hardware and various operating system components that execute outside the kernel in *user space*. In the structure and organization of a typical microkernel system (Figure 1), each system service is cleanly separated from the other modules. This characteristic allows the system to be extended with new services without greatly impacting other modules.

Several abstractions from the Mach microkernel[*] shown in Table 1 were used in IBM's microkernel. By using CMU's Mach 3.0 microkernel as a base for IBM's commercialized effort, IBM expected the microkernel would achieve rapid acceptance because of its familiarity in academic circles. To adapt the academic code for commercial use, IBM made code enhancements to increase robustness and efficiency. In addition, new functionality included in many second generation microkernel systems was added to the microkernel.

- *Synchronous* IPC is used in the IBM microkernel instead of the original asynchronous approach in Mach. This approach significantly reduced the complexity of the code. The *L3*

[*] The PowerPC version of the IBM microkernel 2.0 is this paper's basis for technical description. We did not have access to the Intel-specific version of the source code. Much of the source code is machine independent, however.

Microkernel Objects	Description
Tasks	Units of resource allocation and protection; each task has its own address space.
Threads	Basic computational entities or units of CPU utilization.
Messages	Packets of data used to communicate in the kernel, between user tasks, or between different kernel subsystems.
Ports	Protected communication channels between tasks.
Memory objects	Objects in a task's address space; forms the basis of the virtual memory system.

Table 1: Microkernel Objects and Definitions

OS microkernel[13] also implements this model.

- IBM introduced a *thread migration* model to the microkernel. Thread migration operating systems include OSF/1 MK 7.1, Sun's Spring Operating System [7], and University of Utah's migrating threads version of Mach[5].
- IBM added *power management* support to the microkernel. Although Mach and OSF/1 MK have processor idling options, neither have a devoted power management service that developers can use to design power saving routines. This service is essential for battery powered devices.
- IBM utilizes direct user-to-user data copy support. Instead of the two copies required in Mach, the IBM microkernel requires only one copy. With the synchronous IPC, message buffering in the kernel is unnecessary, and data can be transferred directly when kernel interpretation is unneeded. The *L3* Operating System [10] also supports direct mapped copies.
- The IBM microkernel supports embedded processors that have smaller memories and compact kernel requirements. The Chorus Company offers customizable operating system products for a wide range of embedded applications, as well.
- Logical clocks and timers are implemented in the IBM microkernel to support real-time processes. Time adjustments are made to logical clocks instead of physical clocks.
- Security was improved in the IBM microkernel by adding support for port restricted rights and security tokens. These mechanisms can reduce the number of security checks during RPC.

Design Overview of the IBM Microkernel

IBM envisioned that the microkernel would unify operating systems platforms within the company and run on a wide array of processors ranging from embedded systems to massively parallel architectures. IBM planned to limit the number of microkernel components; this reduced the microkernel's size, complexity, and maximized the amount of code that ran outside the kernel as application programs. Many important components of Workplace OS were outside of the microkernel including support for multiple operating system *personalities*. Personalities were planned OS/2, DOS, and UNIX. IBM planned to encourage OEMs to develop additional personalities that would operate in user-space.

IBM also envisioned that the microkernel would support embedded systems that do not possess large amounts of memory or secondary storage, such as PDAs and set-top boxes. To reduce the memory footprint for this class of product, subsetting, specialized libraries, and compression were used. Specifically, the embedded microkernel uses a different task manager, device manager, does not include the default pager, uses a subset of the name services, uses a different set of device drivers, and uses a different run-time library. For memory management, a portion of the *embedded microkernel* executable code was stored compressed in ROM thus reducing the microkernel's size significantly. Compression was performed when the ROM image was built, while decompression was on-demand. Embedded microkernel memory management is described further in the VM section.

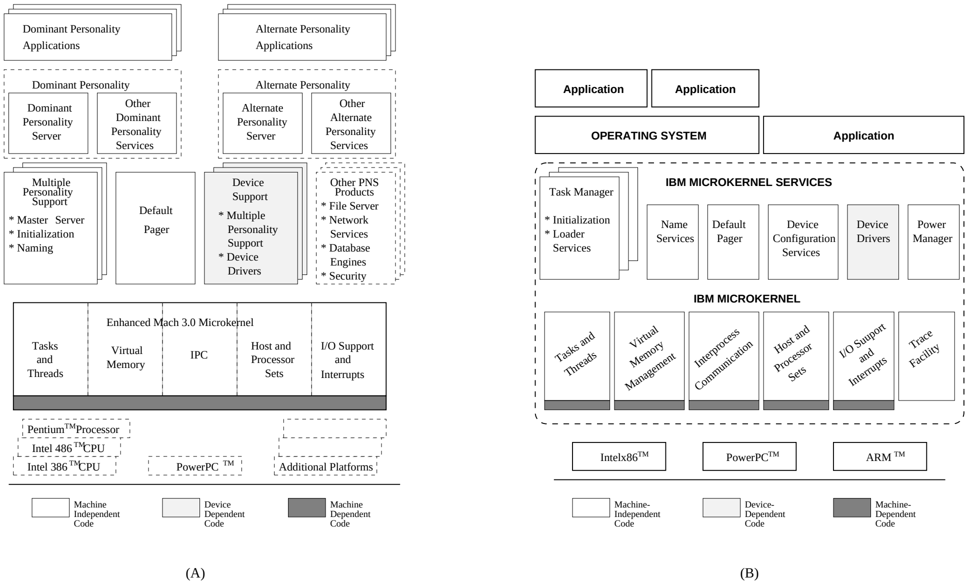

Figure 2: (a) The envisioned IBM microkernel structure, (b) The delivered IBM microkernel

Figure 2 shows the structures of the envisioned and the delivered microkernel. Although not all of the modules were completed in the planned microkernel, most of the essential ones were. The essential difference between the Figure 2(a) and Figure 2(b) concerns the lack of support for multiple operating systems. In both (a) and (b) the operating system is located outside the microkernel and uses the services at layers below. The delivered system was split into three major components: Personalities, Microkernel Services, and Microkernel. Shared library support was also provided.

Operating System Personalities

The envisioned microkernel Figure 2(a) shows support for multiple personalities. The *dominant personality* provides the primary operating system visible to the user. The *alternate personalities* were designed to provide additional operating system services to the user concurrently. Personalities supply two essential components: the *application program interface* (or API) and the *application binary interface* (or ABI). The API provides the set of functions that the personality exports, and thus the API contracts what functionality the personality implements. The ABI dictates the format of the binaries that the personality executes. A UNIX personality need only support the POSIX API. However, an AIX personality must execute native AIX binaries in addition to supporting the AIX API.

There are three reasons why personality support was important to IBM. First, DOS, Windows, and Mac programs are currently the dominant software on store shelves. Obtaining shelf space for a new incompatible software variant would be difficult, even for IBM. Second, users expect not to discard significant investments in software applications, no matter how impressive the new operating system promises to be. Third, compatibility for the PowerPC was essential in order for users to adopt it. The Power Personal System Division's success required that software products provide ABI compatibility. Personalities protect the user's software investment in addition to providing a necessary and familiar API.

The delivered microkernel shown in Figure 2(b) shows support for only one operating system. As we will see later, personality development was one of the most significant problems in Workplace OS.

Microkernel Services

Many personalities have similar functional components. IBM envisioned a generic version of these common components could be designed for use in a shared manner. IBM calls the shared services that are common to several operating systems *Personality Neutral Services* (PNSs). PNSs shown in Figure 2(a) reduce the size and complexity of each personality, and maximize the amount of shared application code. Networking, file systems, scheduling, paging and security services are examples of PNSs. In some systems such as Windows-NT it would be quite hard to extend the operating system scheduler to support real time scheduling. In Workplace OS, a real-time scheduler could be easily substituted as a PNS component in user space.

The delivered microkernel shown in Figure 2(b) provides no PNSs, although there were shared services. Important components in the delivered Microkernel Services are:

Task Manager This service is responsible for initialization and loader services. After initialization is complete, control is transferred to the task manager which loads the paging, naming, and other programs that execute early in the initialization sequence. The loader can be left active to load additional programs or to attach new libraries.

Name Services Name Services enables clients to locate service providers or to register as a service provider. This service maintains a hierarchy of named objects and contains information to contact servers, access system devices, or locate system data objects.

Device Configuration Services This service is responsible for locating and managing all I/O related hardware visible to the system software. It determines which I/O hardware is active and grants access to it.

Power Management The microkernel provides APIs for user programs to control devices that can be powered down. With this facility, one can construct programs which save power. Since the power management system controls devices rather than allowing programs to control devices directly, the facility also makes management more secure.

Microkernel

Both the envisioned and delivered microkernel have identical components, except that the delivered microkernel contains a trace facility. The base components consist of:

Interprocess Communication The IPC system allows threads running in different tasks to communicate. The system supports reliable delivery of synchronous messages on ports. The sender must be prepared to block until a receiver is found and the data is transferred to the destination task.

Task and Thread Management The IBM microkernel manages the program execution environment and the scheduling of the threads. A task consists of an address space and a collection of threads that execute in that address space, as in Mach.

Virtual Memory Management The virtual memory component provides support for large, paged, sparse address spaces composed of memory objects. The IBM microkernel manages memory protection and sharing of memory objects. Address spaces are managed by mapping or allocating memory objects within them. *Copy-on-write* is used when a new address space is created to inherit memory objects and avoid copying pages. Only when a program updates pages are copies made.

Others *I/O support, Host and Processor support*, and a *Trace Facility* are also supported in the microkernel. The I/O support provides access to I/O resources. Host and Processor support provides boot services, clock services, processor management, and scheduling services. The trace information generated by the Trace Facility assists debugging, problem determination, and performance analysis.

Shared Libraries

An executable's run-time environment is a function of the underlying threads package, the OS personality, and the services employed. Because the IBM microkernel provides multiple execution environments, the ABI of the platform and the behavior and requirements of these services are of particular concern to a shared library designer. Usually shared libraries are explicitly selected and linked with the executable. These libraries will require execution environment-dependent services or execution environment-independent services. The execution environment-dependent services include routines to create threads, terminate threads, and obtain thread IDs. A dynamically linked library (DLL), or shared library, can be linked to tasks running in different execution environments if the library does not directly use any of the execution environment-dependent services. A shared library is called a *common shared library* (CSL) if it satisfies these requirements.

IBM considered ways to support separate independent and dependent code portions for DLLs. One approach was to use a different version of the execution environment-dependent library for each different execution environment. At link time, proper versions of the execution-environment-dependent shared library could be linked. However, this approach was not adopted because it would present substantial management difficulties. Instead, IBM chose to provide an Operating System Abstraction (OSA) layer for libraries that provide support for multiple execution environments. OSA functions provide execution-environment-dependent services (threading package services) to shared libraries. When the library is loaded, the OSA functions are initialized for the proper execution environment in which the task is running.

IPC Management

One of the most significant changes IBM made to the microkernel was in the IPC subsystem. The IBM microkernel supports *interprocess communication* (IPC) optimization by restricting interactions to *synchronous* IPC. This style of communication requires the sending thread receive a reply from the recipient before it can proceed. Interactions are restricted to using one-way *messaging* or synchronous two-way *remote procedure calling* (RPC). As observed by Cheriton [2], the complexity of queueing multiple messages is eliminated using synchronous IPC, since each message requires an explicit response before it can proceed. Receives need only save one message at a time and the elimination of lengthy message queues reduces complexity. In addition, the in-microkernel RPC[8] means: 1) reply ports will not be needed by either the RPC client or server code, and 2) reply message formats can be tightly coupled with request message formats. The effect of the former is simplification of message formats. The effect of the latter is that user-level code can avoid performing message format verifications for each reply, thus improving performance.

To support the tight coupling of request message formats with replies, the IBM microkernel uses *portclasses* and *signatures.* Specifically, every port is created as an instance of a portclass. The portclass defines the format of messages that can be transferred through ports of a particular class. This permits a small constrained set of message formats to be pre-registered with the IPC system once and avoids the Mach overheads in verifying message format correctness during each message transfer. Nonetheless, portclasses do not handle dynamically created message types, which IBM claimed required excessive overhead to process. Therefore, to handle message formats that are not known at portclass creation time, a portclass can be defined so that message formats are presented at transmission time. The microkernel associates a *signature* with each request and reply message. The signature defines how the data is to be interpreted and passed by the IPC primitives; signatures are not allowed to be changed after the port is created. This allows senders and receivers to check the registered message formats once and then confidently use the port without concerns of message format changes. IBM claimed this optimization reduced the amount of overhead associated with runtime checking during RPC, since reply message formats can be tightly coupled with request message formats using the same signature.

Adoption of synchronous microkernel-supported RPC eliminated some port right difficulties with two one-way communications (logical RPCs) that occurred in Mach 2.5 and 3.0. The microkernel provides notifications to inform clients and servers when changes in port-rights occur such as deletion of a port, when a receive right for a port is destroyed, or when the last send or send-restricted right for a port is deallocated. Problems between clients and servers arose when a client opened a connection to a server and the client specified a reply port to communicate on. That reply port was used once during initial connection setup and usually discarded. However, logical RPCs did not leave a client failure-isolated from the server. Servers had to receive and discard many unwanted port-deleted notifications during normal client shutdown because the server stored a copy of the initial reply port used to establish the connection. Further, clients could not rely on the server's integrity. Because of this, an errant server could flood a client that made an initial connection to it earlier with many messages on its initial reply-port. Mach 3.0 solves these problems using a *send-once right.* A client that makes a logical RPC call allocates a reply port, holds the receive right to it, and sends that right in a message to the server as a send-once right. The server replies to the send-once right, removing the right immediately from its name space. Since the right is immediately destroyed once the reply occurs, this eliminates unnecessary notifications when the port is destroyed later[18].

Because Mach did not support a microkernel-based RPC, reply ports had to be supplied by the user-level code and there was additional overhead, in each direction, for port name lookup and management. In addition, there was overhead to track which reply ports were associated with which threads. On the other hand, IBM's microkernel-supported RPC maintains the reply port inside the microkernel, stored in the client thread structure. This permits IBM to eliminate the *send-once right*, since the microkernel manages replies through direct thread linkages where the client thread and server thread are bound through the client's reply port. Also, microkernel-supported RPC obviates the need to insert the user-supplied reply port in the server. Instead, when the reply message is sent across the reply port, the server loses its internally-managed send right to it automatically. Thus, synchronous IPC considerably simplifies port management and code complexity.

In addition to the basic send and receive rights adopted from Mach 3.0, IBM has added a *send-restricted right.* This modified send right is identical to the normal send right except that this right is prevented from circulating between tasks whose security levels are not equal. Since IBM hoped agencies such as ARPA or DOD would adopt the microkernel, this right could be used to implement multi-level security policies or to improve RPC performance. Security mechanisms are described further in the Security Section.

Thread Optimization for RPC and Scheduling

The IBM microkernel supports multi-threaded programming with support for both user-level threads and kernel-level threads. The microkernel itself uses kernel-level threads to provide internal support functions such as page eviction, thread reclamation, and scheduler priority computations. The *C-threads* run-time library [8] is used to manage user-level threads. The implementation of *C-threads* calls microkernel thread functions when C-threads are created or destroyed. The IBM implementation of C-threads supports preemptive scheduling and parallel execution.

IBM devoted considerable attention to improving threads support. The goal was to avoid many performance penalties usually associated with microkernel systems. One source of penalties arises from locating the operating system server in user space and using a single, general-purpose IPC message mechanism to implement RPC. This adds significant overhead to the system call path and makes OS access to kernel or user memory costly[3]. However, when programs shared the same address space, it is possible to replace IPC message-based RPC with a form of direct procedure call known as *short-circuited RPC*[3]. Short-circuited RPC is many times more efficient than message passing and provides significant performance improvements.

IBM adopted a version of the Utah migrating threads mechanism to support short circuiting. Threads are divided into two separate entities called the *thread body* and *shuttle.* The thread body represents the execution context of a computation and retains all resource information. The shuttle comprises the schedulable entity, meaning it holds the priority and the resource accounting attributes. Under normal circumstances the roles of the thread body and the shuttle are not visible. However, when RPC is invoked, the two entities are used. Specifically, a client thread gives up its shuttle to a server thread which claims the shuttle. Figure 3 illustrates thread migration.

IBM's changes to the thread structure were designed to improve RPC performance by reducing the overhead associated with context switches for a typical RPC call. Specifically, in the original Mach 3.0 static thread implementation, a thread performing an RPC required two context switches. The client thread needed to be swapped out and replaced by the server thread, then

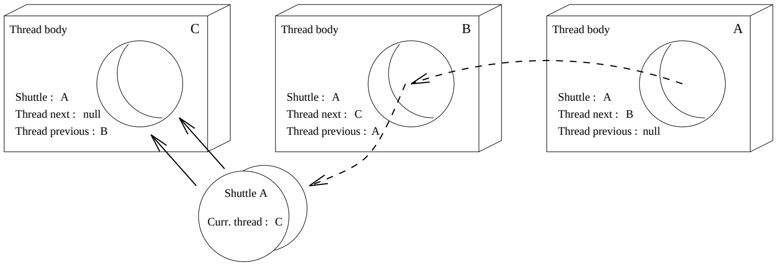

Figure 3: Thread Migration process

swapped in again. Using the migrating thread model [5] a full context switch is unnecessary since thread information is embedded in shuttles. The scheduling information associated with the client is formally handed off to the server for the duration of the RPC. Only the address space and some subset of the CPU registers needs to be switched. In addition, rescheduling is not required since there is no need to search for higher priority work in the system because the client thread's priority is used. A factor of 3 to 4 performance improvement was reported when using this technique in the Utah work[5].

For scheduling, IBM supports a range of policies for environments including multiprocessors, workstations, and PDAs. The microkernel implements the mechanisms to manage the physical processors, where each processor is a member of a *processor set*. The processor set schedules a subset of the threads using one uniform scheduling policy. Each thread has an associated scheduling policy inherited from its containing task which governs how the processor set schedules the threads.

Numerous scheduling policies support the wide range of environments and architectures that the designers envisioned for the microkernel. The policies include the First-In-First-Out (FIFO), Round-Robin, and a Timeshare scheduling policy. The PDA scheduler uses a very simple thread scheduling policy that uses 256 priorities[12]. A numeric priority is statically assigned to each thread and the highest priority thread is allocated the CPU. However, problems were anticipated between different application designers fighting for superior performance. Some application writers could chose thread priorities that monopolize the PDA. IBM stated that the solution to this problem was not a technical one, but a business one. Unfairness from specific problem applications would be readily apparent and the marketplace would reject monopolistic software.

Power Management

The IBM microkernel provides a Power Management Framework to manage devices that are sensitive to power concerns. The Framework matches the physical organization of the underlying hardware. The hardware itself is managed through the manipulation of the power management model represented in the Framework. Programmers can implement specific power management strategies for hardware devices using this Framework.

Figure 4 shows the relevant objects in the power management architecture as defined by IBM[9]. The power management *policy object* provides the implementation of a specific power management policy. This policy describes all the rules for managing power among all the different hardware devices that the policy controls. The relationship between a policy object and its controlling policy is established through a specific interface. If a policy object is not attached to a specific policy, no events can be sent from that object to the controlling policy. *Managed objects* represent physical components in the system and describe various topological constructs used to construct a model of a computer's power system. Five managed objects are of particular concern: power envelopes, event sources, power objects, power consumers, and power suppliers. *Power envelopes* shown in Figure 4 represent the power domains in a system; they describe units that consume and supply power. Envelopes are used to organize the topological constructs so they match the physical system layout. *Event sources*, also shown, notify the power management system of important changes that should be relayed to the power policy. For example, the spinning down of a disk is an event of interest to the power policy. *Power objects* represent the power attributes of physical components such as devices, as shown in Figure 4. Typically, there is one power object for each device. Lastly, *power suppliers* represent components such as batteries, power supplies, and UPSs and *power consumers* represent components such as LCD displays, modems, and processors.

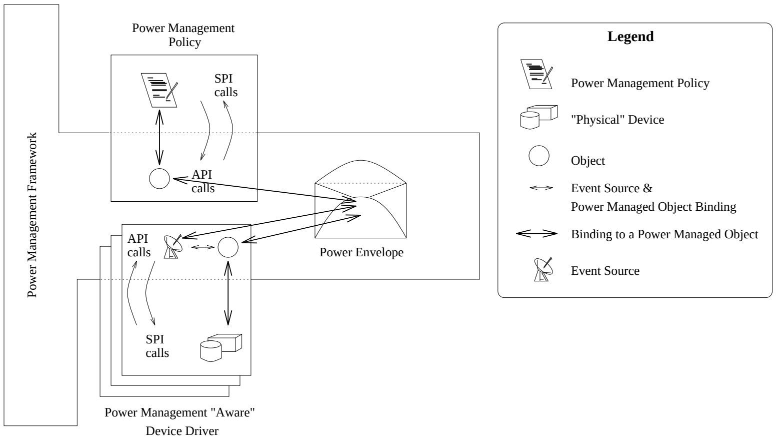

Figure 4: Power Management System Components

A typical device driver can be made *power aware* by integrating the device driver into the Framework. To convert a conventional device into a "power aware" device, each device driver must support a power management object. That object must be attached to the Framework through a power envelope. Figure 4 shows one (or more) power management object(s) attached to a power envelope and one power management policy attached to the same envelope. Typically the envelope will contain several event sources and several power managed objects. Policies must attach themselves to envelopes in this Framework. When the policy is attached to an envelope, the policy can manage the power for all objects in the envelope. There is one policy associated with each envelope and thus one uniform policy may control several devices. Once the policy (object) is attached it can receive messages that contain event notifications.

Events are generated by power management objects and directed to *event sources* instead of directly sending events to policy objects. These event sources are part of the power envelope and are explicitly created. The types of events that can be generated from event sources are predetermined and fixed; event types cannot be changed. An object may have several event sources associated with it, but an event source is associated with only one event. The event source sends the event onto the policy or it may delay the event for a predetermined period of time. Event sources can be masked to prevent events from being issued even if an event occurs. Both event sources and power managed objects must be bound to an envelope explicitly.

The Framework uses *Service Provider Interfaces* (SPIs) as a means for the Framework to call external routines provided by the policies or the object implementations. Two types of SPI calls can be made: Object SPI calls or Policy SPI calls. For example, SPI calls from the Framework to the policy (Policy SPI calls) can notify a policy of an event. In return, the policy reacts to the events with a set of predefined actions and makes calls into the Framework using the API. Typical events that may be of interest to a policy include the state of a device being powered on or off, a device's screen dimming or brightening, or a disk spinning up or down.

Call Type	**API**	**SPI**	**Call Type**[*]	**API**
Generic Object	6	4	Conduit	3
Event Source	6	6	Envelope	3
Object with States	13	12	Event Generation	14
Policy	8	19	Platform	6
Power Object	2	1	Framework	15
Transition	4	2	Representative	2
State	4	2		
Total	43	46	**Total**	43

Table 2: Number of System calls for Power Management System

Table 2 reflects the considerable effort IBM spent on the power management system. Over 130 new calls were added in 11,437 lines of code. IBM also produced a 450 page Power Management user manual that describes the functions, parameters, message flows, and return values for power management in more detail[9].

Virtual Memory Management

IBM enhanced the PowerPC version of the Virtual Memory (VM) subsystem with: direct user-to-user copies, an improved default memory manager, and embedded system support[†]. For the first, the PowerPC uses the *block address translation table* (BAT)[‡] to support direct message transfers from the sender's address space to the receiver's address space. The PowerPC BAT maps a large block of data into pinned, contiguous real memory. A "cross-mapped" address is established for the BAT to transfer the data using mapping techniques rather than copying. To transfer the data the target address space must "cross map" the sender's data addresses into its own page maps. The cross mapped address must be a power of 2, it must be 128KB to 8MB, both the starting and effective addresses must be a multiple of the range, and for the PowerPC 601, the effective address range cannot overlap any of the effective address ranges being used for other cross-mapped operations. If these constraints are met, and the message needs no translation, the microkernel can reduce the number of message copies to one copy[§].

[*] There are no SPIs for these calls.

[†] We did not have the Intel source code to confirm similar support for the Intel platform, but documentation indicates similar support is available on the Intel.

[‡] This is specific to the PowerPC.

[§] The cross-mapped address table is stored in the thread structure to assure its state is preserved in case a page fault occurs.

For the second, the VM default memory manager manages temporary non-persistent memory objects. Memory backed by the default pager is called *anonymous* memory. Anonymous memory has no named port associated with it and is stored in a cache of physical memory. To manage the anonymous memory more effectively, IBM improved the implementation to use four queues instead of the three used in Mach. These queues are called the *inactive, active, free,* and *prefault queue.* The new prefault queue is used to store prefetched pages from disk; this optimization returns the page more quickly than a traditional fault which must wait for the page to be returned.

IBM used care when adding the new prefault queue. Because the prefault queue consumes memory, the VM module was adapted to support situations when memory is low. Specifically, the IBM microkernel cannot count on external pagers to free memory since external pagers are non-privileged and may be stalled waiting for free memory. Instead, the code was adapted to reclaim memory from the prefault queue prior to reclaiming memory from the inactive queue. Additionally, pages are carefully dribbled from the active queue to the inactive queue to assure that pages on the inactive queue get a second chance to be referenced before they are reclaimed. Care in flow control guarantees that pagers keep pace with microkernel operations. In addition, when memory is very low, pages are not paged out to disk since paging operations consume memory.

For the third, the embedded microkernel VM does not include the default pager found in the base microkernel since there is no paging or conventional secondary storage as in workstations. Instead a *compression (CMP) memory manager* is used that stores components of the microkernel in compressed form. For performance reasons, frequently referenced performance-critical sections of the microkernel are not compressed and execute in-place. Nonetheless, IBM states that roughly 50% of the embedded Microkernel can be compressed[8]. For example, user tasks and microkernel service tasks, in addition to a number of kernel services, can be compressed. IBM also planned a Microkernel File server with a *reduced footprint* FAT file system. A reduced footprint server would have fewer APIs and less functionality for embedded systems.

When a page fault occurs in the embedded microkernel, CMP memory manager routines are invoked. The CMP pager acquires a page frame in RAM and decompresses the ROM code into the RAM memory. This decompression is performed dynamically as the ROM pages are accessed. Once a page is decompressed, accesses to that page are diverted to the new page in RAM. This new page is marked *defective.* Defective pages contain addresses that have no meaning because they have been moved without internal addresses having been translated. Several functions called from the fault handler perform *patching* which fixes the defective pages. The patching converts absolute addresses in ROM pages, RAM pages, or virtual addresses, to absolute addresses in RAM pages. The lack of translation hardware in embedded systems requires this manual software translation process.

Lastly, for the microkernel programmer using the VM subsystem, the IBM microkernel adds functions that allow users to provide hints to the page eviction and prefetch algorithms and to provide paging advice concerning the expected amount of reads, writes, and inactivity for certain memory ranges. These hints can improve performance.

Logical Clocks for Real-Time Support

IBM's goals for the microkernel included support for real time processing. Nontheless, real-time application support can be a challenging issue since real-time applications have substantially different performance and operational criteria than conventional operating systems applications. For example, the microkernel must support preemptibility, provisions for adequate real time process scheduling, main memory scheduling, and cache scheduling. Whereas conventional performance criteria emphasizes average performance, real time applications must bound worst-case performance. Some of these issues require new policy modules that can be outside of the microkernel. For example, scheduling policies and memory management can be embedded in policy modules that are support by the microkernel but external to it in handlers or policy modules. These modules can be easily changed to support new policies without changing the microkernel itself. Thus, IBM expected few actual changes to the microkernel for real time support and expected to use one base microkernel for real-time and non real-time applications. Nonetheless, we do focus on the one main feature that IBM added to the base microkernel to support real time processing: *logical clocks*.

In many systems, problems arise when users adjust the time because certain clock properties must always hold. First, the values returned by the physical clock must be monotonically nondecreasing. Although the offset may be a negative value (to preserve the time continuity), the rate can never be less than 0. Second, both relative time[*] and absolute time[†] are restricted to positive values. If there are any real-time processes executing on a processor, care must be used when modifying the physical clock or problems may arise. Nonetheless, users may wish to adjust the time because: *(a)* an initial setting of the clock may be needed, *(b)* the clock may need to more accurately reflect the passage of real time, and *(c)* political time changes may be needed (daylight savings, leap seconds, etc.).

IBM introduces the *logical clock* object to support real-time processing in the microkernel. Logical clocks abstract modifications to the physical clock by preventing direct access to it. Only logical clocks can be read and modified. Logical clocks smooth the effect of abrupt time corrections that result from forward and backward corrections by clock synchronization protocols. For example, since more than one logical clock is allowed, it is beneficial to use one logical clock directly in the synchronization protocol but to read time from another logical clock that amortizes the adjustments made to the first clock. This prevents time from appearing to run backward.

Logical clocks can be used to support *timers*. *Alarms* are used to inform a thread of a predefined periodic passage of time as measured by a timer. The alarm can either be synchronous or asynchronous with respect to the execution of the target thread. For *synchronous alarms*, only one alarm can be set per thread. For *periodic asynchronous alarms*, a timer is created explicitly and a *timer port* is returned.

Security

The IBM microkernel provides security using 1) traditional tasks and *port rights*, 2) security tokens, and 3) restricted port rights. Tasks support memory objects that can be mapped into the virtual memory address range with specific protection rights. When machine instructions attempt to access specific address ranges, access permission is verified. Since ports represent services, port rights are used to protect access to services. A task's port rights are increased as additional port rights are returned from RPCs or by third parties that insert new port rights locally. Task *special ports* are used to represent initial privileges to services such as a host port, bootstrap port, or kernel task port.

[*] an interval time whose starting time is the current time

[†] a point of time relative to the origin of the clock

Security tokens identify a *subject* in the security system. Security credentials are presented to authorize subjects to gain access to resources. For example, a typical security service allocates security tokens and maintains a mapping between security tokens and security context information. Each security subject may have several tasks that need to work on its behalf. Tokens are used to identify subjects with a *credential.* Three special types of token are: *task-security tokens* which identify the task making a request, *thread-proxy-token* which identifies the privilege of the service requester,[‡] and the *task-proxy-token* which manipulates the thread-proxy-token values for the task's threads.

At the request of the server, the sender's security token can be attached in an unforgeable manner to every message that the sender directs to the server. Servers can use the token to contact an authorization service to convert it into a set of credentials which can be compared against the actions being requested. The IBM design allows the token-to-credentials relationship to be cached in the server and for a special form of *send-right* to represent a pre-approved authority to perform certain actions. For example, if a message arrives on a port for which only *restricted-send-rights* have been created, the server assumes that the actions are authorized since restricted-send-rights can only be moved or copied to a task with the same task-security-token. Without restricted-send-rights, send-rights could be propagated in an unrestricted manner, requiring the server to repeatedly verify the sender.

An example may illuminate how these security mechanisms can be used. Consider a patient who fills a prescription at the local pharmacy. If a physician prescribes a drug and this written prescription is taken into the pharmacy, the pharmacy must take the time to contact the physician to verify the prescription's authenticity. Also, the pharmacy must assure that the patient's health plan permits the specific drugs. This action is similar to the message authentication when there is a security token attached. However, the nurse could 1) call in each prescription on a secure phone line and the pharmacy could have a "caller-id" box and 2) the nurse could verify the patient's health insurance. These actions save time for the pharmacy since authentication and permission are not needed. In this latter case, the pharmacy merely dispenses the medicine assured the prescription is authentic and allowed.

If a patient selects a pharmacy that is out of medicine, the pharmacy may help the patient locate a nearby pharmacy that can fill the prescription. In this example, the prescription can be authenticated at the pharmacy but not filled. The pharmacy can establish a secure channel to transfer the prescription to another location. This is similar to each pharmacist possessing a task-security-token and using restricted-send-rights. Only registered pharmacists may communicate in this manner, obviating the need for authentication. Indeed, restricted-send-rights have the property that they can be moved or copied freely between entities with the same security token. This can save the patient time and provide for faster location of needed medicines.

In summary, with restricted-send-rights a server need only validate the client once. Thereafter, because the subject's credentials have been checked, and because restricted-send-rights can only be transferred between tasks with identical task-security-tokens, strong guarantees can be made concerning the validity of the request. Restricted-send-rights can increase server performance because the burden to check credentials can be performed once and only once.

[‡] This token is used when a server wishes to take on the identity of one of its clients

Evaluation

In this section we examine the quality of IBM source code. We compared Workplace software to Mach 3.0[*] using various measurement programs. Aside from the normal readability measures, both Halstead's metrics and McCabe's cyclomatic complexity metric were used on the source codes[1, 4]. Halstead's theory of software science is probably the best known and most thoroughly studied composite measures of software complexity. *McCabe's Cyclomatic Complexity* measures control flow complexity.

Our analysis was performed with a measuring package named *C Metrics*[†]. The package was used to measure Purity Ratio, Volume, Effort, and Cyclomatic Complexity. In particular, *purity ratio* is a measure of the code optimization: a measure greater than 1.0 indicates more optimized code. *Volume* measures a program's size using the quantity of operators and operands as parameters. *Effort* reflects the number of *mental discriminations* a programmer performs to write a program. Each mental comparison consists of a number of elementary mental discriminations and indicates a measure of program difficulty. Cyclomatic Complexity measures control flow in a procedure based on the number of decision statements. The program *C_COUNT ver7.0*[‡] was used to count code lines[§], comment lines[¶] and *grep* was used to count the number of assert statements. In addition, we constructed a program to count the number of lines per procedure, the number of procedures or functions, and then computed the averages. Table 3 gives the results.

	MACH			**IBM**					
	IPC	VM	KERN	IPC	VM	KERN	CLOCK	PWRM	TRACE
Code	12089	9149	13799	10142	23755	16901	6386	11437	3068
Cmnt	4629	4961	5375	7221	10762	8257	2768	3776	937
Cmnt/Code	0.382	0.542	0.389	0.711	0.453	0.488	0.433	0.330	0.262
Procs	239	173	464	142	310	444	206	327	90
Ln/Proc	50.6	52.9	29.7	71.4	76.6	38.1	31.0	35.0	34.1
Asserts	718	94	77	414	412	322	11	0	9
Purity	1.38	1.38	2.08	1.77	1.45	2.04	1.69	2.34	0.83
Volume	358739	255005	449972	263177	756480	574271	203687	263227	73153
Effort ($\times 10^6$)	111	56	310	98	376	382	61	130	3
CC	7.4	5.7	3.7	4.8	5.8	3.4	3.6	2.8	3.2

Table 3: Static Code Evaluation Results.

[*] Available via ftp at mach.cs.cmu.edu in /src/mkernel.

[†] Available via ftp at ftp.wustl.edu in /languages/c/unix-c/utils.

[‡] Available via ftp at sunsite.unc.edu in /pub/Linux/devel/lang/c.

[§] Including lines for preprocessor.

[¶] Code lines consist of lines containing code and preprocessor instructions. Comment lines consist of comments on separate lines or comments appearing on the same line as source code. Copyrights and RCS histories were not counted in the total comment lines.

Measure	**Formula**	**Comments**
Purity Ratio	$PR = \hat{N}/N$	A measure of code optimization. A low ratio suggests that there is excessive code. The higher the ratio above 1.00, the greater the optimization in the code. Classes of *impurities* include complementary operations, ambiguous operands, synonymous operands, common subexpressions, unwarranted assignment, and unfactored expressions.
Volume	$V = N \times \log_2 n$	An appropriate measure of program "size". Since the size metric should not reflect the number of characters used, it is expressed in bits. This length depends only on the number of elements in the vocabulary.
Effort	$E = V/\hat{L}$ $\hat{L} = 2/n_1 \times n_2/N_2$	Reflects the number of mental discriminations required to reduce a preconceived algorithm to an actual implementation. This measure seems to correlate highly with experience.
Cyclomatic complexity	$V(g) = e - n + 2$	Measures control flow in the function based on the number of decision statements in the code. Extended cyclomatic complexity includes both decision-making statements and decision-making predicates.

Legend : n_1 = number of unique operators; n_2 = number of unique operands; n = number of unique operators and operands; N_1 = total number of operators; N_2 = total number of operands; N = length as determined by $N = N_1 + N_2$; $\hat{N}$ = predicted length as determined by $(n_1 \times \log_2 n_1) + (n_2 \times \log_2 n_2)$.

Table 4: Definitions for Halstead's and McCabe's metrics.[4]

The results from these measurements provided significant insights. The IPC module of the IBM microkernel was reduced in size by approximately 16%, nearly 100 procedures or functions were eliminated, and the ratio of comments to code increased 33%. This is consistent with the assertion that the complexity of the IPC was reduced by using synchronous IPC and indicates careful coding practices were used in the IPC module. The purity ratio of the IPC module increased by .39, meaning the code was optimized when IBM changed the IPC system. The comparison of cylomatic complexities of the IPC modules shows that the code was less complex than before. In addition, Figure 5 also confirms a reduction in the number of IPC functions. However, the number of procedures with more than 140 lines of code was not significantly reduced by the IBM effort.

The IBM VM module which includes new RPC specific routines, an expanded default memory manager, and support for the embedded microkernel has more than doubled in code size. The ratio of comments to code decreased by 9%, the number of lines of code per procedure increased by roughly 30% on average, and over 120 new procedures were created. We find this particularly disturbing because VM code is often the most challenging to understand and good commenting, of small, well-decomposed functions is essential. Figure 5 shows this increase in number of functions, and more disturbingly, the number of functions greater than 140 lines. The increase in the volume of code supports the claim that this module was vastly expanded. The purity ratio of the VM code increased by a small margin; however, the effort value for the IBM VM module was increased 6 times than the original. In the VM code, there were many places where the code was replicated, with only slight variations. Several mismatched commenting styles were observed in the pmap module.

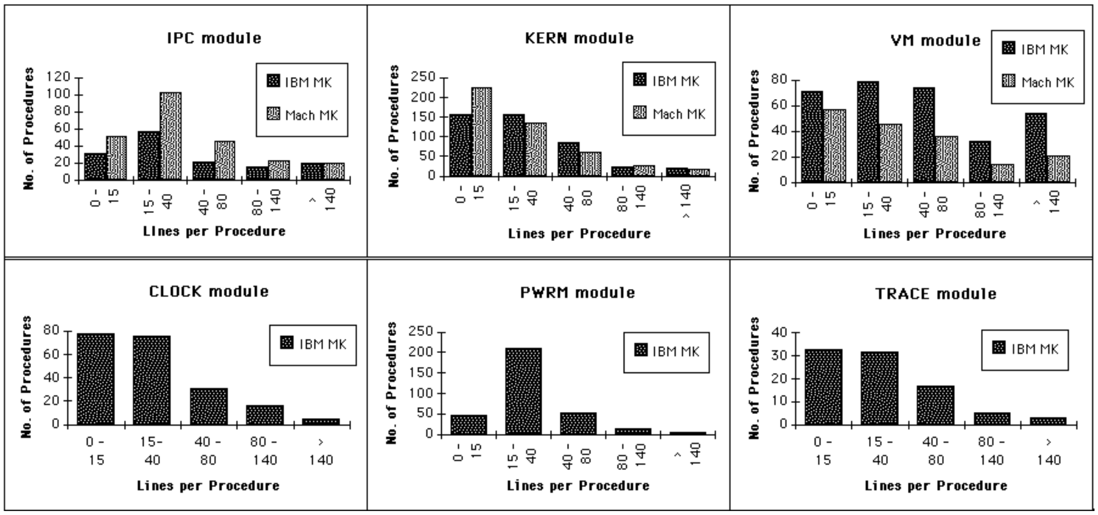

Figure 5: Size of microkernel procedures.

The KERN module increased in code size by 20% with a proportional increase in comments. IBM eliminated 20 procedures or functions. However, of the various modules we examined, this module appears the least changed statistically. The modules CLOCK, PWRM and TRACE add 623 new procedures or functions averaging roughly 31 lines of code per function. The purity ratio, module volume, effort, and cyclomatic complexity differed marginally from Mach's.

The CLOCK and PWRM modules appear to have highly optimized code. In fact, PWRM has the highest purity ratio of all the modules and also the lowest cyclomatic complexity, leading us to believe that this module was planned and implemented carefully. On the other hand, the TRACE module possesses the lowest purity ratio even if it has the lowest effort value.

The number of assert statements in Workplace is also revealing since asserts can greatly improve the reliability of the code and help locate bugs. In the IBM microkernel, the total number of assert statements increased by roughly 30% but the total number of lines of code doubled. In IBM's CLOCK, PWRM, and TRACE modules, there are only 20 assert statements in over 20,000 lines of newly developed IBM code. IBM marketed the microkernel as a more robust implementation of Mach and we find the lack of asserts in the newly developed code disturbing.

Overall, the average number of lines of code per procedure increased by a minimum of 20% in all the IBM code. IBM significantly increased the size of code by 73% and also increased the comment ratio by 4%. In IPC, VM, and KERN only 20 more procedures/functions were added despite the reduction in functions in the IPC module. However, in these modules IBM added over 15,700 lines of new code. For the most part, this indicates that IBM enhanced existing functions rather than devising new functions in these modules. IBM often split a single function into two functions, in order to allow the new function to be called from some other function. However, new features like the compression cmp_pager for the embedded microkernel, required new code be devised.

Table 5(a) provides execution performance of the IBM microkernel on the IBM PowerPC

Test	**Time (μs)**
Null RPC	13.89
Thread Create	55.72
Thread Terminate	34.28
Task Create	107.46
Task Terminate	63.84
VM Allocate	12.11
VM Deallocate	7.48

System	**CPU, MHz**	**$\sim$MIPs**	**μs**	**$\sim$MIPs$\times\mu$s**
L3	486, 50	10	10	100
LRPC	FF-CVAX	2	157	314
QNX	486, 33	6.6	76	502
IBM MK	PPC 604, 100	60	14	840
Amoeba	68020, 15	1.5	800	1200
Mach	486, 50	10	230	2300
Dash	68020, 15	1.5	1920	2880

Table 5: (a)Performance Evaluation Results. (b)Null RPC comparisons.

604/100MHz. We could not perform multi-machine testing of the microkernel because the delivered code did not work with the on-board Ethernet chip set supplied with the PowerPCs.[*] Thus, only single site performance measurements are presented. These measurements include time to perform a null RPC, create and terminate tasks and threads, and time to allocate and deallocate memory[†]. In Table 5(b) only the IBM MK was measured directly; the remaining performance data taken from Liedtke for comparison purposes[10]. These results show that the IBM microkernel has better RPC performance than Amoeba, Mach, and Dash. IBM did not surpass the speed of L3, LRPC, or QNX in RPC tests.

Workplace OS Project History

To reiterate, the basic premise of Workplace OS was: 1) IBM would adopt and improve the CMU Mach 3.0 microkernel for use on PDAs, the desktop, workstations, and massively parallel machines, and 2) that several operating system personalities would execute on the microkernel platform concurrently. This strategy required machine independence, multiple personalities, and concurrent operation of personalities. In the previous sections of this paper, we focused on the microkernel. We now turn our attention to project history and issues in personality development.

In January 1991 the project was conceived. The first presentation of IBM's new operating systems strategy was given to internal management with a chart referred to as the Grand Unification Theory of Operating Systems (or GUTS, for short). GUTS outlined how one microkernel would unify several operating systems with common "subsystems". At the end of 1991, a small team from Boca Raton, Florida and Austin, Texas had been formed to begin work on a version of the Mach Microkernel to support OS/2, the lead personality. In the summer of 1992, the prototype was underway and there was good progress. IBM successfully demoed OS/2, DOS, DOS/Windows, and Unix running on the Mach microkernel at the Fall Comdex in 1992.

Soon after, IBM announced plans to develop OS/2, DOS, and Unix as microkernel personalities for both PowerPC and Intel architectures[17]. Internal discussion at IBM focused on AIX. Finally, at Comdex in 1993, IBM Chairman Louis Gerstner announced that the microkernel would not

[*] IBM acknowledged this difficulty and suggested we purchase additional Ethernet boards. We ordered these boards after providing precise part numbers, receiving them, and installing them. However, when we installed them, the microkernel still did not communicate on the network.

[†] These are single page vm_allocate/vm_deallocate tests.

replace AIX. IBM realized that many AIX users would not accept performance penalties associated with microkernel systems. IBM was also concerned with the microkernel presenting a competitive impediment against high performance HP or Sun Unix systems than ran directly on the hardware. Instead, Gerstner told AIX customers that they would be able to migrate to Workplace OS, later if they were interested.

Intense Workplace development followed. IBM divided five major personality projects across separate divisions. Each division was required to support their own OS personality on the microkernel. Reports indicated there were over 400 microkernel programmers[15] and 1500 OS/2 programmers[16] geographically distributed in different divisions working on Workplace. In addition, a microkernel business unit was established to market the microkernel and create University relationships. Further, in conjunction with microkernel development, IBM planned to offer workstations based on the Motorola PowerPC which was touted as a more economical RISC machine that would execute personalities compatible with the Intel processor. A Power Personal Systems Division was established with development facilities in Austin, Texas, Boca Raton, Florida and Yamato, Japan. The Division defined the PowerPC systems standard and planned to sell systems that ran all personalities. The PReP (PowerPC Reference Platform) specification was created to specify the structure of the components for PowerPC machines[15]. In addition, IBM planned to push for acceptance of the microkernel as a new standard through the OSF Research Institute where many of the microkernel enhancement ideas originated.

In May 1994 the division director of RISC Systems software announced plans to study an AIX personality for Workplace. A small internal research team of less than ten members was assembled and led by an IBM Research Fellow. The press announcement included information that indicated a significant problem with development of the AIX personality was that of byte-ordering. IBM reminded customers that monolithic AIX runs perfectly well on the PowerPC and that IBM needed time to address this difficult endian problem.

IBM was silent on the issue of AIX for approximately seven months. However, in January 1995, IBM announced the AIX personality effort would be halted and an AIX personality for Workplace would not be built. Instead in February, IBM announced that it would offer a non-AIX personality for Workplace. The new UNIX personality was meant for users that might otherwise find themselves rebooting between microkernel OS/2 and AIX. However, this effort was not well received and later abandoned.

In October 1995 IBM finally announced the general availability of Version 1 of the microkernel for the PowerPC. In the first year of release, IBM had several commercial vendors and Universities that adopted the microkernel including Digital Equipment Corporation, LG Electronics (Goldstar), Komatsu, Trusted Information Systems, and Bell-Northern Research. In addition, Universities such as Carnegie-Mellon University, Notre Dame, Oregon Graduate Institute, University of California at Irvine and Riverside, University of Texas, Arlington, Helsinki University of Technology, Tokyo University, and Cornell University were using IBM's microkernel for their research.

Later in October, media reports began to circulate that the PowerPC 620, which was the basis for the new improved desktop PowerPcs, was bug-ridden. Shortly after this news, IBM cancelled the Workplace project and folded the Power Personal Division. The latest and last release of the microkernel, version 2.0, was distributed to microkernel adopters early the following year. The final release supported the Motorola PowerPC, Intel x86, and the ARM(*Advanced RISC Machine*) embedded processor. Approximately one year after cancellation of Workplace OS, the IBM Boca Raton, Florida facility was closed.

Table 6 summarizes events or assumptions versus outcomes associated with IBM microkernel

Personality Assumptions or Events	Personality Outcomes
UNIX, OS/2, OS/400, Windows would run side-by-side on the microkernel as personalities.	OS/2 and Windows-NT ported to PowerPC without IBM microkernel.
PowerPC price/performance would attract customers along with a multi-personality operating environment.	Delays in introduction of software and hardware reduced performance advantage of PowerPC. Without personalities, PowerPC incompatibilities outweigh benefits.
IBM invites Apple to adopt the microkernel for Mac OS.	Apple refuses microkernel adoption and states that the microkernel has excessive resource requirements. About one month later IBM announces a marketing study indicating there is no customer demand for Mac OS on the PowerPC.
IBM announces a study of a Workplace AIX personality. Although OS/2 and the microkernel were Little Endian and AIX was Big Endian, IBM would address this by assigning an IBM Fellow to "crack the problem" along with IBM's best research minds.	AIX personality abandoned in January 1995 and IBM denies any original plans to support an AIX personality. IBM cited success of monolithic AIX on PowerPC and continued work on OS/2. Privately, some IBM executives admitted Workplace was dead.
In February 1995 IBM announces non-AIX personality for the microkernel described by IBM as a variant of AIX with a non-AIX API.	A new version of UNIX is not welcomed. The media expresses concerns whether Workplace will be successful.
In March 1995 IBM clings to late summer release date for OS/2.	In June 1995 IBM ships PowerPCs with monolithic AIX and Windows-NT. In July 1995 IBM quietly announces it will have Mac OS on PowerPCs next year.
In October 1995 reports circulate that the PowerPC 620 is bug-ridden.	IBM announces end of Power Personal Division and the end of the microkernel strategy. One year later IBM Boca Raton was closed permanently.

Table 6: Assumptions/Events and Outcomes in Personality Development

personalities. The most significant problems with the Workplace project concerned underestimating the difficulty of implementing personalities, rather than issues associated with microkernel development.

Observations and Lessons

Experience can be a good teacher in the design and analysis of computer systems. We can learn a great deal from engineering failures, perhaps more than from successes. We offer the following postmortem of Workplace based on our experience and intuition concerning the design of large software systems:

1. IBM underestimated the difficulty in creating personalities. Each personality required extensive restructuring to support shared PNSs. These divisions were not always easy to delineate or implement as common subsystems. PNSs require that personality designers communicate effectively to reach common agreement on goals and implementation strategies for shared services.[*] Also, co-existence of ABI-compatible personalities with different "endian-ness" presented insurmountable problems. Thus, personalities would fail to emerge and generalize for concurrent use on the microkernel. It was easier to create a strategic business plan for the financial markets rather than a working operating system with multiple, cooperating personalities.

2. The IBM microkernel suffered from the "second-system effect"[6] where the second system is embellished with frill after frill. The general tendency is to over-design the second system and to propose overly ambitious and generalized functions.

3. IBM considered personalities late in the project as compared to the microkernel where it placed considerable effort on functionality, efficiency, and portability early in the design. In contrast, in Windows-NT, personalities were considered early in the design and there was no emphasis on generalizing the NT microkernel to all products.

4. Liedtke[11] argues that microkernels are hardware dependent similar to optimizing code generators. He argues that not only the coding, but even the algorithms used inside a microkernel and its internal concepts are extremely processor dependent. Workplace's failure supports the claim that a new processor may require a new microkernel design. However, the success in building Version 2.0 which runs on the Intel, PowerPC, and ARM processor, provides partial refutation.

5. It was poor judgment for IBM to require all divisions to support the microkernel until more research had been conducted on its applicability across the diverse product lines, the applicability across existing software products, and to have one prototype with all *essential* personalities. In addition, IBM should have marketed personality-based PowerPCs *after* having the essential personalities prototyped. Associating the success of Power Personals with the success of personality development was unwise.

6. Sound software engineering practices and more effective management may have improved coordination and allowed divisions better responsiveness in creating the product. PNSs require that personality designers communicate effectively to reach agreement on delineations. Effective management is needed to reach compromises and manage coordination of these efforts.

Conclusion

The IBM microkernel project resembled the IBM/360 project. Both were designed for a family of computers spanning the range from small machines to large scientific computers. Only one set of software was envisioned for these systems and this aspect was supposed to reduce the

[*] Brooks [6] p.16-19 has a good discussion of the problems that arise on large software projects when there is a need to communicate between parties.

Feature	Summary Evaluation
IPC	Simplified code & improved performance using synchronous IPC.
Threads	Performance-enhanced for RPC; adapted to support flexible scheduling policies.
Power Management	Complete power subsystem added with APIs and SPIs; power sensitive devices managed through the Framework.
Virtual Memory	Direct user-to-user copy support; improved paging policies; support for embedded systems with small memories.
Logical Clock	Support for real-time systems.
Code Evaluation	Average lines per procedure or function 20% larger; too few asserts in new code; VM more complex; better RPC performance than Mach.
Security	Better security and authorization mechanisms; improved efficiency for fast RPC.

Table 7: Workplace Component Evaluation

maintenance problems for IBM and allow users to move programs and applications freely from one IBM system to another. The IBM/360 tried to be all things for all people[14] and as a result, did none of its tasks especially well. The system was written in assembly language by thousands of programmers, resulting in millions of lines of code. The microkernel involved hundreds of programmers that produced thousands of high-level instructions that never made their way into a significant commercial product.

The failure of Workplace OS can be attributed to 1) failure of personalities to generalize beyond their tested previous scope and 2) overly grandiose ambitions for using the microkernel on all IBM products. In isolation, Workplace microkernel components seemed well considered and reasonably carefully designed as shown in Table 7; IBM worked closely with OSF Research Institute during the design. Indeed, the final version of the microkernel, Version 2.0, operated on PowerPCs, Intel machines, and the ARM processor. However, when the components were combined, Workplace was an overly embellished system. In the final analysis, the failure of Workplace can be attributed to the lack of personalities and a vision for the microkernel that suffered from the second-system effect. This led IBM to one of the largest operating system failures in modern times.

REFERENCES

1. B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New York, 2nd edition, 1990.
2. D. R. Cheriton. *The Thoth System: Multi-Process Structuring and Partability.* Elsevier, North Holland, 1982.
3. M. Condict, D. Bolinger, E. McManus, D. Mitchell, and S. Lewontin. Microkernel modularity with integrated kernel performance. Technical report, Open Software Foundation Research Institute, Cambridge, Mass, 1994.
4. Thomas Drake. Measuring software quality: A case study. *Computer*, 29(11):78–87, November 1996.
5. Bryan Ford and Jay Lepreau. Evolving mach 3.0 to a migrating thread model. In *USENIX Conference Proceedings*, pages 97–114, San Francisco, CA, Winter 1994. USENIX.
6. Jr. Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, 1982.

7. Graham Hamilton and Panos Kougiouris. The spring nucleus: A microkernel for objects. In *USENIX Conference Proceedings*, pages 147–159, Cincinnati, OH, Summer 1993. USENIX.
8. IBM. IBM microkernel: Overview and Programming Guide Release 2.0, March 1996.
9. IBM. IBM microkernel: Power Management Programming Reference Release 2.0, March 1996.
10. Jochen Liedtke. Improving ipc by kernel design. In *14th ACM Symposium on Operating System Principles (SOSP)*, pages 175–188, Asheville, NC, December 1993. ACM.
11. Jochen Liedtke. On microkernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain, CO, December 1995. ACM.
12. Larry Loucks, Ravi Manikundalam, and Freeman Rawson III. A microkernel-based operating system for personal digital assistants. In *Proceedings of the Fourth Workshop on Workstation Operating Systems.* IEEE, October 1993.
13. Marion Schalm, Jean Wolter, and Michael Hohmuth. L3 documentation, December 1995.
14. A. Silberschatz and P. Galvin. *Operating System Concepts.* Addison-Wesley Publishing Company, Inc., 1995.
15. Tom Thompson and Bob Ryan. Apple, IBM Bring PowerPC To the DeskTop, April 1994.
16. New York Times. IBM Set To Introduce its Latest Os/2 Software, September 25, 1996.
17. Unknown. Windows NT and Workplace OS:plug it in. *Byte Magazine*, 19(1):166, January 1994.
18. Uresh Vahalia. *UNIX Internals: The New Frontiers.* Prentice Hall, Upper Saddle River, New Jersey 07458, 1996.